

# Building an AI Agent in TypeScript

Build a real, running agent loop: a deterministic, explicitly simulated decision step genuinely calling the real MCP tools from "Building an MCP Server."

**Intermediate** 30 min

Generated on 5 October 2026

Read online: <https://www.learnforgex.com/en/topics/building-an-ai-agent>

---

## Building an AI Agent in TypeScript

Every lesson in this category so far has been conceptual: "What Is an AI Agent?" defined the observe-decide-act loop, "Agent Planning and Reasoning Patterns" went deeper into how the "decide" step can work, and "Controlling Agent Behavior" covered the guardrails a loop like that needs. This lesson builds one -- a real, running agent loop, reusing the exact `GeoFactsServer.ts` MCP server from "Building an MCP Server", and everything below was actually written, compiled, and run to confirm it behaves exactly as shown.

This lesson does not call a real LLM API. Read "What's Real and What's Simulated in This Lesson" before continuing -- it explains exactly what that means and why, so the rest of the lesson isn't misread as claiming more than it does.

### What We'll Build

1. Reuse `GeoFactsServer.ts` from "Building an MCP Server" unchanged -- the same `get_capital_city` and `calculate_sum` tools.
2. Add `AgentLoop.ts` : a decision step and a real agent loop built around it.
3. Add `RunAgentDemo.ts` : wiring that connects a real MCP client to the server and runs the loop against a real goal.
4. Install, compile, and run all of it, and look at the real, step-by-step trace it produces.

### What's Real and What's Simulated in This Lesson

This lesson's entire point is to make the agent loop from "What Is an AI Agent?" concrete with real, running code -- but doing that honestly means being explicit about one thing: this lesson has no access to a real LLM API, so it cannot call one. Here is exactly what that means for the code below:

- **Real:** the agent loop itself -- the observe-decide-act cycle from "The Agent Loop: Observe, Decide, Act" -- genuinely runs, step by step, in `AgentLoop.ts`.
- **Real:** every tool call. `AgentLoop.ts` calls the real MCP Client's `callTool()`, which sends a real `tools/call` message (exactly as "From Concepts to Wire Format: JSON-RPC 2.0" described) to the real `GeoFactsServer.ts` server, which genuinely executes `get_capital_city` or `calculate_sum` and returns a real result.
- **Real:** the step limit -- the loop is genuinely bounded, exactly as "Step and Iteration Limits" in "Controlling Agent Behavior" described.
- **Simulated:** the *decision* step. A real agent uses a language model to read the goal and decide, through open-ended reasoning, what to do next. This lesson replaces that with `decideNextAction()`: a small, fully deterministic, hand-written function that recognizes two fixed patterns of text ("capital of X" and "sum of ...") and nothing else. It is explicitly labeled `(simulated)` in its own output, both in code comments and in what it prints -- it is not a language model, does not "understand" the goal the way a model would, and must not be read as a demonstration of real LLM behavior.

The reason for building it this way: the goal of this lesson is to show the agent loop -> tool call -> MCP server -> tool result -> agent loop flow genuinely working, using the real MCP infrastructure from "Building an MCP Server" -- not to demonstrate real LLM API integration, which "What Would Change With a Real Model" at the end of this lesson covers conceptually instead.

## Project Setup (Reusing Building an MCP Server)

This lesson uses the exact same project setup as "Building an MCP Server": the same `package.json` (with `"type": "module"` and dependencies on `@modelcontextprotocol/sdk` and `zod`) and the exact same `tsconfig.json` (`"module": "NodeNext"`, `"moduleResolution": "NodeNext"`, `outDir: "dist"`). If you still have that project folder, you can add this lesson's three files directly into it. If not, recreate `package.json` and `tsconfig.json` exactly as shown in "Create package.json" and "Create tsconfig.json" in "Building an MCP Server", then run `npm install` once before continuing.

```
ai-agent-demo/
├─ package.json
├─ tsconfig.json
├─ GeoFactsServer.ts
├─ AgentLoop.ts
└─ RunAgentDemo.ts
```

## Reusing the Server: GeoFactsServer.ts

`GeoFactsServer.ts` is copied here completely unchanged from "Building an MCP Server" -- the same `get_capital_city` and `calculate_sum` tools, the same `CAPITALS` lookup (France, Japan, Turkey), the same `isError: true` behavior for an unknown country:

```

// A small, self-contained MCP server exposing two tools: one that looks up
// information the model would not otherwise know (get_capital_city), and one
// that performs a computation (calculate_sum). Kept in its own file/function
// so it can be reused both by a real stdio-connected server process and,
// as in RunServerWithClient.ts, by an in-process demo client.
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { z } from "zod";

const CAPITALS: Record<string, string> = {
  France: "Paris",
  Japan: "Tokyo",
  Turkey: "Ankara",
};

export function createGeoFactsServer(): McpServer {
  const server = new McpServer({ name: "geo-facts-server", version: "1.0.0" });

  server.registerTool(
    "get_capital_city",
    {
      title: "Get Capital City",
      description: "Looks up the capital city of a given country.",
      inputSchema: { country: z.string().describe("Country name, e.g. 'France'") },
    },
    async ({ country }) => {
      const capital = CAPITALS[country];
      if (!capital) {
        return {
          content: [{ type: "text", text: `No capital known for "${country}"` }],
          isError: true,
        };
      }
      return { content: [{ type: "text", text: `The capital of ${country} is ${capital}.` } ] };
    }
  );

  server.registerTool(
    "calculate_sum",
    {
      title: "Calculate Sum",
      description: "Adds a list of numbers together.",
      inputSchema: { numbers: z.array(z.number()).describe("Numbers to add") },
    },
    async ({ numbers }) => {
      const total = numbers.reduce((acc, n) => acc + n, 0);
      return { content: [{ type: "text", text: `Sum: ${total}` } ] };
    }
  );

  return server;
}

```

Nothing about this file is agent-specific. That's deliberate -- an MCP server has no idea whether the client calling it is a single tool call, as in "Building an MCP Server", or a full agent loop, as in this lesson. The tool definitions, their schemas, and their behavior stay

completely unchanged either way, exactly as "From This Demo to a Real Deployment" noted about transports -- here it's the *caller* that changes, not the server.

## The Agent Loop: AgentLoop.ts

AgentLoop.ts contains both halves described in "What's Real and What's Simulated in This Lesson": the simulated `decideNextAction()` function, and the real `runAgentLoop()` function built around it.

```
// This file implements a genuine agent loop: a decision step chooses an
// action, that action is really executed against a real MCP server (through
// a real MCP client's callTool -- the exact same call RunServerWithClient.ts
// in "Building an MCP Server" used), and the real result is fed back into
// the next decision step. None of the tool-calling or MCP communication
// below is faked.
//
// What IS faked, on purpose, is the decision step itself. A real agent uses
// a large language model to read the goal and the conversation so far and
// decide, through open-ended reasoning, what to do next. This lesson has no
// access to a real LLM API, so decideNextAction() below is a small, fully
// deterministic, hand-written function: it recognizes two fixed patterns of
// text ("capital of X" and "sum of ...") and nothing else. It is NOT a
// language model, does not "understand" the goal in any general sense, and
// must never be mistaken for real model behavior -- it exists only so the
// *loop* around it (decide -> act -> observe -> repeat) can be demonstrated
// with a real, running, inspectable agent, calling the real
// get_capital_city and calculate_sum tools from GeoFactsServer.ts.
import type { Client } from "@modelcontextprotocol/sdk/client/index.js";

export interface AgentStep {
  thought: string;
  toolCall?: { name: string; arguments: Record<string, unknown> };
  toolResult?: string;
  toolIsError?: boolean;
}

export interface AgentResult {
  steps: AgentStep[];
  finalAnswer: string;
  stoppedByStepLimit: boolean;
}

interface Decision {
  thought: string;
  toolCall?: { name: string; arguments: Record<string, unknown> };
  finalAnswer?: string;
}

// (Simulated model.) A deterministic, explicitly hand-written stand-in for
// what a language model would do at each turn: look at the goal and what
// has happened so far, and decide on ONE next action. Real agents replace
// this single function with an LLM call -- runAgentLoop() below, and
// everything it does with the result, stays exactly the same either way.
function decideNextAction(goal: string, history: AgentStep[]): Decision {
  const alreadyCalled = (toolName: string) =>
```

```

    history.some((step) => step.toolCall?.name === toolName);

    const countryMatch = goal.match(/capital of (\w+)/i);
    if (countryMatch && !alreadyCalled("get_capital_city")) {
      const country = countryMatch[1];
      return {
        thought: `(simulated) Goal asks for the capital of "${country}". Plan: call
get_capital_city.`,
        toolCall: { name: "get_capital_city", arguments: { country } },
      };
    }

    const sumMatch = goal.match(/sum of ([0-9, and]+[0-9])/i);
    if (sumMatch && !alreadyCalled("calculate_sum")) {
      const numbers = sumMatch[1]
        .split(/,|and/i)
        .map((part) => part.trim())
        .filter((part) => part.length > 0)
        .map(Number);
      return {
        thought: `(simulated) Goal asks for the sum of [${numbers.join(", ")}]. Plan: call
calculate_sum.`,
        toolCall: { name: "calculate_sum", arguments: { numbers } },
      };
    }

    // Both sub-goals this demo cares about are already satisfied (or never
    // matched at all) -- (simulated) combine what the real tool calls
    // returned into a final answer instead of calling anything else.
    const facts = history
      .filter((step) => step.toolResult !== undefined)
      .map((step) => step.toolResult)
      .join(" ");
    return {
      thought: "(simulated) All sub-goals are covered by prior tool results. Plan: answer
using them.",
      finalAnswer: facts.length > 0 ? facts : "(no tool results were gathered to answer
from)",
    };
  }

  // The real agent loop: decide, act for real, observe the real result,
  // repeat -- bounded by maxSteps so a decision function that never resolves
  // to a final answer cannot run forever (see "Controlling Agent Behavior"
  // for why a step-limit guardrail like this matters in general).
  export async function runAgentLoop(
    client: Client,
    goal: string,
    maxSteps: number
  ): Promise<AgentResult> {
    const steps: AgentStep[] = [];

    for (let i = 0; i < maxSteps; i++) {
      const decision = decideNextAction(goal, steps);

      if (decision.finalAnswer !== undefined) {
        return { steps, finalAnswer: decision.finalAnswer, stoppedByStepLimit: false };
      }
    }
  }

```

```

if (decision.toolCall) {
  const result = await client.callTool({
    name: decision.toolCall.name,
    arguments: decision.toolCall.arguments,
  });
  const content = result.content as Array<{ type: string; text: string }>;
  const text = content.map((c) => (c.type === "text" ? c.text : "")).join(" ").trim();
  steps.push({
    thought: decision.thought,
    toolCall: decision.toolCall,
    toolResult: text,
    toolIsError: Boolean(result.isError),
  });
}

return {
  steps,
  finalAnswer: "(step limit reached before a final answer was produced)",
  stoppedByStepLimit: true,
};
}

```

`decideNextAction()` looks at the goal text and, using nothing more sophisticated than two regular expressions, decides on one of three things: call `get_capital_city` if the goal mentions a capital and that tool hasn't run yet, call `calculate_sum` if the goal mentions a sum and that tool hasn't run yet, or -- once neither condition applies -- combine whatever real tool results have been gathered so far into a final answer. Every `thought` string it produces starts with the literal text `(simulated)`, on purpose, so nothing printed by this demo can be mistaken for genuine model reasoning.

`runAgentLoop()` is the real loop from "The Agent Loop: Observe, Decide, Act": on each of up to `maxSteps` iterations, it calls `decideNextAction()` (observe + decide), and if the decision includes a tool call, it really executes `client.callTool(...)` (act) and records the real result before looping again. If `maxSteps` is reached without a final answer, `stoppedByStepLimit` comes back `true` instead of the loop continuing forever -- this is the step-limit guardrail from "Step and Iteration Limits" in "Controlling Agent Behavior", implemented as an ordinary bounded `for` loop.

## Running the Agent: RunAgentDemo.ts

`RunAgentDemo.ts` connects a real MCP `Client` to `GeoFactsServer.ts` over the same in-memory transport "Building an MCP Server" used, then runs `runAgentLoop()` against a goal that genuinely needs both tools:

```

// Wires a real MCP client and the real GeoFactsServer.ts server together
// over an in-memory transport (the same pattern RunServerWithClient.ts used
// in "Building an MCP Server"), then runs a real agent loop against them.
// The (simulated) decision-making in AgentLoop.ts is clearly labeled at
// every step below -- everything else (the tool calls, the MCP messages,
// the results) is genuinely executed.
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { InMemoryTransport } from "@modelcontextprotocol/sdk/inMemory.js";
import { createGeoFactsServer } from "../GeoFactsServer.js";
import { runAgentLoop } from "../AgentLoop.js";

async function main() {
  const server = createGeoFactsServer();
  const [clientTransport, serverTransport] = InMemoryTransport.createLinkedPair();
  const client = new Client({ name: "agent-demo-client", version: "1.0.0" });

  await Promise.all([
    server.connect(serverTransport),
    client.connect(clientTransport),
  ]);

  const goal = "What is the capital of Japan, and what is the sum of 12, 30, and 8?";
  console.log(`Goal: ${goal}\n`);

  const result = await runAgentLoop(client, goal, 5);

  result.steps.forEach((step, i) => {
    console.log(`Step ${i + 1}: ${step.thought}`);
    if (step.toolCall) {
      console.log(` Tool call:   ${step.toolCall.name}(${JSON.stringify(step.toolCall.arguments)})`);
      console.log(` Tool result: ${step.toolResult}${step.toolIsError ? " (isError: true)" : ""}`);
    }
    console.log();
  });

  console.log(`Final answer: ${result.finalAnswer}`);
  console.log(`Stopped by step limit: ${result.stoppedByStepLimit}`);

  await client.close();
  await server.close();
}

main();

```

The goal -- "What is the capital of Japan, and what is the sum of 12, 30, and 8?" -- was chosen specifically because answering it correctly requires two separate real tool calls, in sequence, with the agent loop deciding on its own to make a second call after the first one's result came back. A `maxSteps` of 5 gives the loop comfortable room above the two steps this goal actually needs, while still keeping the step-limit guardrail from "Controlling Agent Behavior" in place.

## Install, Compile, and Run

With `package.json`, `tsconfig.json`, `GeoFactsServer.ts`, `AgentLoop.ts`, and `RunAgentDemo.ts` all saved in the same project folder, run the same three commands "Install, Compile, and Run" in "Building an MCP Server" used:

```
npm install
npx tsc -p tsconfig.json
node dist/RunAgentDemo.js
```

That third command produces this, exactly:

```
Goal: What is the capital of Japan, and what is the sum of 12, 30, and 8?

Step 1: (simulated) Goal asks for the capital of "Japan". Plan: call get_capital_city.
Tool call:  get_capital_city({"country":"Japan"})
Tool result: The capital of Japan is Tokyo.

Step 2: (simulated) Goal asks for the sum of [12, 30, 8]. Plan: call calculate_sum.
Tool call:  calculate_sum({"numbers":[12,30,8]})
Tool result: Sum: 50

Final answer: The capital of Japan is Tokyo. Sum: 50
Stopped by step limit: false
```

## What Actually Happened

Step 1: `decideNextAction()` matched "capital of Japan" in the goal text and (simulated) decided to call `get_capital_city`. `runAgentLoop()` then really called `client.callTool({ name: "get_capital_city", arguments: { country: "Japan" } })` -- exactly the same MCP `tools/call` message "Connect a Client: RunServerWithClient.ts" showed for the same tool -- and the real server returned "The capital of Japan is Tokyo.".

Step 2: with `get_capital_city` now marked as already called, `decideNextAction()` matched "sum of 12, 30, and 8", extracted the numbers `[12, 30, 8]`, and (simulated) decided to call `calculate_sum`. The real tool call returned "Sum: 50".

Step 3 (which never needed to run as a tool call): with both sub-goals now covered by real tool results in `history`, `decideNextAction()` produced a `finalAnswer` combining them, and the loop returned before reaching `maxSteps`. `stoppedByStepLimit: false` in the output confirms the loop ended because the (simulated) decision step concluded it was done, not because it ran out of room.

## Trying the Error Path

Changing the goal to "What is the capital of Wakanda?" and rerunning produces this real output:

Goal: What is the capital of Wakanda?

Step 1: (simulated) Goal asks for the capital of "Wakanda". Plan: call `get_capital_city`.

Tool call: `get_capital_city({"country": "Wakanda"})`

Tool result: No capital known for "Wakanda". (`isError: true`)

Final answer: No capital known for "Wakanda".

Stopped by step limit: false

This confirms the same `isError: true` behavior "Why the 'Wakanda' Example Exists" in "Building an MCP Server" described still reaches the agent loop correctly: `runAgentLoop()` doesn't crash or treat the error specially -- the real error result becomes part of `history` like any other tool result, and `decideNextAction()`'s (simulated) final-answer step reports it as-is.

## What Would Change With a Real Model

Replacing `decideNextAction()` with a real LLM call is the only change a production version of this agent would need -- `runAgentLoop()`, `GeoFactsServer.ts`, and the MCP wiring in `RunAgentDemo.ts` would stay exactly the same, because none of them depend on how the decision is made. A real implementation would send the model the goal, the tool descriptions (name + description + schema, as "Defining a Tool: Name, Description, and Schema" in "Tools and Function Calling" described), and the history so far, and let the model's own reasoning -- following one of the patterns from "Agent Planning and Reasoning Patterns" -- choose the next action instead of a regular expression. Everything this lesson verified about the loop, the real tool calls, and the step-limit guardrail applies unchanged either way.

## Best Practices

- Keep the decision step and the loop mechanics in separate functions, as `decideNextAction()` and `runAgentLoop()` do here -- see "What Would Change With a Real Model" -- it's what makes replacing a simulated decision step with a real model call a contained, isolated change.
- Label a simulated or mocked component unambiguously, in both code and any output it produces -- see "What's Real and What's Simulated in This Lesson" -- so no one mistakes a deterministic stand-in for real model behavior.
- Enforce a step limit in code, exactly as `runAgentLoop()` does, rather than trusting the decision step to always terminate on its own -- see "Step and Iteration Limits" in "Controlling Agent Behavior".

## Common Mistakes

- **Mistaking this lesson's `decideNextAction()` for a simplified language model.** As "What's Real and What's Simulated in This Lesson" explained, it recognizes two fixed

text patterns and nothing else -- it has none of a real model's generality, and the code and output both mark it (simulated) for exactly this reason.

- **Assuming a real LLM-backed agent needs a different tool-calling mechanism than this demo used.** "What Would Change With a Real Model" showed that `runAgentLoop()`'s real tool-calling code doesn't change at all -- only the decision step does.
- **Skipping the step limit because the decision function "obviously" terminates.** `decideNextAction()` here happens to always terminate in at most three steps, but "Step and Iteration Limits" in "Controlling Agent Behavior" covered why a hard limit still belongs in the loop -- the guardrail matters for decision steps that don't reliably terminate, including real, model-driven ones.

## Summary, Cheat Sheet, and Glossary

### Summary

- This lesson's agent loop, tool calls, MCP communication, and step limit are all genuinely real, running code -- reusing `GeoFactsServer.ts` unchanged from "Building an MCP Server".
- Only the decision step, `decideNextAction()` in `AgentLoop.ts`, is simulated: a small, deterministic, explicitly-labeled stand-in for real model reasoning, recognizing two fixed text patterns.
- `runAgentLoop()` implements "The Agent Loop: Observe, Decide, Act" as an ordinary bounded loop, calling the real MCP `Client`'s `callTool()` for every tool action and recording the real result.
- A goal needing two tool calls (`get_capital_city` then `calculate_sum`) produced a verified, real, step-by-step trace ending in a correct final answer, without hitting the step limit.
- Replacing `decideNextAction()` with a real LLM call is the only change needed to move from this lesson's demo to a production agent -- the loop, tool calls, and guardrail all stay the same.

### Cheat Sheet

- Real: agent loop, `client.callTool()` calls, MCP messages, step limit.
- Simulated: `decideNextAction()` -- deterministic, regex-based, labeled (simulated) everywhere it appears.
- Loop shape: `for` up to `maxSteps` -> decide -> (if tool call) really execute + record result -> (if final answer) return early.
- Files: `GeoFactsServer.ts` (unchanged from "Building an MCP Server"), `AgentLoop.ts` (decision + loop), `RunAgentDemo.ts` (wiring + a goal needing 2 tool calls).
- Run order: `npm install -> npx tsc -p tsconfig.json -> node dist/RunAgentDemo.js`.

### Glossary

- **decideNextAction():** this lesson's simulated decision step -- a deterministic function standing in for a real model's reasoning, clearly labeled as such.
- **runAgentLoop():** this lesson's real agent loop -- observes history, calls `decideNextAction()`, executes real tool calls, and enforces a step limit.
- **Step limit ( `maxSteps` ):** the hard, code-enforced maximum number of decide-act cycles `runAgentLoop()` will run, from "Step and Iteration Limits" in "Controlling Agent Behavior".
- **stoppedByStepLimit:** this lesson's flag distinguishing a loop that ended because the decision step reached a final answer from one that was forced to stop by the step limit.